CHAPTER 12

Data Processing and Analysis

The last several chapters covered the main topics of traditional scientific computing. These topics provide a foundation for most computational work. Starting with this chapter, let's move on to explore data processing and analysis, statistics, and statistical modeling. First, we look at the `Pandas` data analysis library. This library provides convenient data structures for representing series and tables of data and makes it easy to transform, split, merge, and convert data. These are important steps in the process[1] of cleansing raw data into a tidy form suitable for analysis. The Pandas library builds on top of NumPy. It complements it with features that are particularly useful when handling data, such as labeled indexing, hierarchical indices, data alignment for comparison and merging of datasets, handling of missing data, and much more. As such, the `pandas` library has become a de facto standard library for high-level data processing in Python, especially for statistics applications. The `Pandas` library contains only limited support for statistical modeling (namely, linear regression). Other packages are available for more involved statistical analysis and modeling, such as `statsmodels`, `patsy`, and `scikit-learn`, which are covered in later chapters. However, for statistical modeling with these packages, `Pandas` can still be used for data representation and preparation. The `Pandas` library is, therefore, a key component in the software stack for data analysis with Python.

Pandas The `Pandas` library is a framework for data processing and analysis in Python. At the time of writing, the most recent version of Pandas is 2.3.1. For more information about the `Pandas` library, and its official documentation, see the project's website at `http://pandas.pydata.org`.

The primary focus of this chapter is to introduce basic features and usage of the `Pandas` library. Toward the end of the chapter, we briefly explore the statistical visualization library Seaborn, which is built on top of Matplotlib. This library provides quick and convenient visualization of data represented as a `Pandas` data structure (or NumPy arrays). Visualization is an essential part of exploratory data analysis, and the Pandas library itself also provides functions for basic data visualization (which also builds on top of Matplotlib). The Seaborn library takes this further by providing additional statistical graphing capabilities and improved styling: the Seaborn library is notable for generating good-looking graphics using default settings.

Seaborn The Seaborn library is a visualization library for statistical graphics. It builds on Matplotlib and provides easy-to-use functions for common statistical graphs. At the time of writing, the most recent version of Seaborn is 0.12.0. For more information about Seaborn and its official documentation, see `https://seaborn.pydata.org/index.html`.

[1] Also known as data munging or data wrangling

R. Johansson, *Numerical Python*, https://doi.org/10.1007/979-8-8688-0413-7_12

Importing Modules

This chapter mainly works with the pandas module, which is imported under the name pd.

```
In [1]: import pandas as pd
```

We also require NumPy and Matplotlib, which are imported in the following way.

```
In [2]: import numpy as np
In [3]: import matplotlib.pyplot as plt
```

For a more aesthetically pleasing appearance of Matplotlib figures produced by the pandas library, select a suitable style for statistical graphs using the mpl.style.use function.

```
In [4]: import matplotlib as mpl
   ...: mpl.style.use('ggplot')
```

Later in this chapter, we use the seaborn module, which we import under the name sns.

```
In [5]: import seaborn as sns
```

Introduction to Pandas

The main focus of this chapter is the pandas library for data analysis. The pandas library mainly provides data structures and methods for representing and manipulating data. The two primary data structures in Pandas are the Series and DataFrame objects, which represent data series and tabular data, respectively. Both objects have an index for accessing elements or rows in the data the object represents. By default, the indices are integers starting from zero, like NumPy arrays, but it is also possible to use any sequence of identifiers as an index.

Series

The merit of indexing a data series with labels rather than integers is apparent even in the simplest examples. Consider the following construction of a Series object. Give the constructor a list of integers to create a Series object representing the provided data. Displaying the object in IPython reveals the data of the Series object together with the corresponding indices.

```
In [6]: s = pd.Series([909976, 8615246, 2872086, 2273305])
In [7]: s
Out[7]: 0     909976
        1    8615246
        2    2872086
        3    2273305
        dtype: int64
```

The resulting object is a Series instance with the data type (dtype) int64, and the elements are indexed by the integers 0, 1, 2, and 3. Using the index and values attributes, we can extract the underlying data for the index and the values stored in the series.

```
In [8]: s.index
Out[8]: RangeIndex(start=0, stop=4, step=1)
In [9]: s.values
Out[9]: array([ 909976, 8615246, 2872086, 2273305], dtype=int64)
```

While using integer-indexed arrays or data series is a fully functional representation of the data, it is not descriptive. For example, if the data represents the population of four European capitals, it is convenient and descriptive to use the city names as indices rather than integers. With a `Series` object, this is possible, and we can assign the `index` attribute of a `Series` object to a list with new indices to accomplish this. We can also set the `name` attribute of the `Series` object to give it a descriptive name.

```
In [10]: s.index = ["Stockholm", "London", "Rome", "Paris"]
In [11]: s.name = "Population"
In [12]: s
Out[12]: Stockholm     909976
         London       8615246
         Rome         2872086
         Paris        2273305
         Name: Population, dtype: int64
```

It is now immediately obvious what the data represents. Alternatively, we can set the index and name attributes through keyword arguments to the Series object when created.

```
In [13]: s = pd.Series([909976, 8615246, 2872086, 2273305], name="Population",
    ...:               index=["Stockholm", "London", "Rome", "Paris"])
```

While it is perfectly possible to store the data for the populations of these cities directly in a NumPy array, even in this simple example, it is much clearer what the data represent when the data points are indexed with meaningful labels. The benefits of bringing the description of the data closer to the data are even more significant when the complexity of the dataset increases.

We can access elements in a `Series` by indexing with the corresponding index (label) or directly through an attribute with the same name as the index (if the index label is a valid Python symbol name).

```
In [14]: s["London"]
Out[14]: 8615246
In [15]: s.Stockholm
Out[15]: 909976
```

Indexing a `Series` object with a list of indices gives a new `Series` object with a subset of the original data (corresponding to the provided list of indices).

```
In [16]: s[["Paris", "Rome"]]
Out[16]: Paris    2273305
         Rome     2872086
         Name: Population, dtype: int64
```

With a data series represented as a `Series` object, we can easily compute its descriptive statistics using the `Series` methods `count` (the number of data points), `median` (calculate the median), `mean` (calculate the mean value), `std` (calculate the standard deviation), `min` and `max` (minimum and maximum values), and the `quantile` (for calculating quantiles).

```
In [17]: s.median(), s.mean(), s.std()
Out[17]: (2572695.5, 3667653.25, 3399048.5005155364)
In [18]: s.min(), s.max()
Out[18]: (909976, 8615246)
In [19]: s.quantile(q=0.25), s.quantile(q=0.5), s.quantile(q=0.75)
Out[19]: (1932472.75, 2572695.5, 4307876.0)
```

All the preceding data are combined in the output of the `describe` method, which provides a summary of the data represented by a `Series` object.

```
In [20]: s.describe()
Out[20]: count          4.000000
         mean     3667653.250000
         std      3399048.500516
         min       909976.000000
         25%      1932472.750000
         50%      2572695.500000
         75%      4307876.000000
         max      8615246.000000
         Name: Population, dtype: float64
```

Using the `plot` method, we can quickly and easily produce graphs that visualize the data in a `Series` object (see Figure 12-1). The `pandas` library uses Matplotlib for plotting, and we can optionally pass a Matplotlib `Axes` instance to the `plot` method via the `ax` argument. The type of the graph is specified using the `kind` argument (valid options are `line`, `hist`, `bar`, `barh`, `box`, `kde`, `density`, `area`, and `pie`).

```
In [21]: fig, axes = plt.subplots(1, 4, figsize=(12, 3))
    ...: s.plot(ax=axes[0], kind='line', title='line')
    ...: s.plot(ax=axes[1], kind='bar', title='bar')
    ...: s.plot(ax=axes[2], kind='box', title='box')
    ...: s.plot(ax=axes[3], kind='pie', title='pie')
```

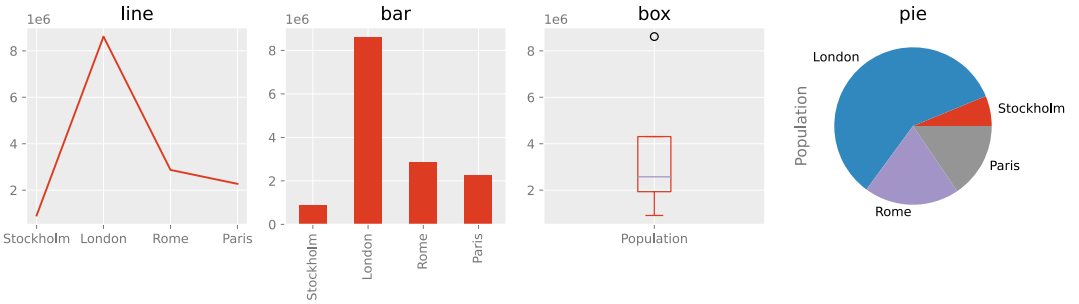

Figure 12-1. *Examples of plot styles that can be produced with Pandas using the* `Series.plot` *method*

DataFrame

As shown in previous examples, a `pandas Series` object provides a convenient container for one-dimensional arrays, which can use descriptive labels for the elements and offers quick access to descriptive statistics and visualization. For higher-dimensional arrays (mainly two-dimensional arrays or tables), the corresponding data structure is the Pandas `DataFrame` object. It can be viewed as a collection of `Series` objects with a common index.

There are numerous ways to initialize a `DataFrame`. For simple examples, the easiest way is to pass a nested Python list or dictionary to the constructor of the `DataFrame` object. For example, consider an extension of the dataset used in the previous section, where, in addition to the population of each city, we also include a column that specifies which state each city belongs to. We can create the corresponding `DataFrame` object in the following way.

```
In [22]: df = pd.DataFrame([[909976, "Sweden"],
    ...:                    [8615246, "United Kingdom"],
    ...:                    [2872086, "Italy"],
    ...:                    [2273305, "France"]])
In [23]: df
Out[23]:
```

	0	1
0	909976	Sweden
1	8615246	United Kingdom
2	2872086	Italy
3	2273305	France

The result is a tabular data structure with rows and columns. Like with a `Series` object, we can use labeled indexing for rows by assigning a sequence of labels to the `index` attribute, and, in addition, we can set the `columns` attribute to a sequence of labels for the columns.

```
In [24]: df.index = ["Stockholm", "London", "Rome", "Paris"]
In [25]: df.columns = ["Population", "State"]
In [26]: df
Out[26]:
```

	Population	State
Stockholm	909976	Sweden
London	8615246	United Kingdom
Rome	2872086	Italy
Paris	2273305	France

The `index` and `columns` attributes can also be set using the corresponding keyword arguments to the `DataFrame` object when it is created.

```
In [27]: df = pd.DataFrame([[909976, "Sweden"],
    ...:                    [8615246, "United Kingdom"],
    ...:                    [2872086, "Italy"],
    ...:                    [2273305, "France"]],
    ...:                   index=["Stockholm", "London", "Rome", "Paris"],
    ...:                   columns=["Population", "State"])
```

An alternative way to create the same data frame, which is sometimes more convenient, is to pass a dictionary with column titles as keys and column data as values.

```
In [28]: df = pd.DataFrame(
    ...:     {"Population": [909976, 8615246, 2872086, 2273305],
    ...:     "State": ["Sweden", "United Kingdom", "Italy", "France"]},
    ...:     index=["Stockholm", "London", "Rome", "Paris"])
```

As before, the underlying data in a `DataFrame` can be obtained as a NumPy array using the `values` attribute and the index and column arrays through the `index` and `columns` attributes, respectively. Each column in a data frame can be accessed using the column name as an attribute (or by indexing with the column label, e.g., `df["Population"]`).

```
In [29]: df.Population
Out[29]: Stockholm     909976
         London       8615246
         Rome         2872086
         Paris        2273305
         Name: Population, dtype: int64
```

The result of extracting a column from a `DataFrame` is a new `Series` object, which we can process and manipulate with the methods discussed in the previous section. Rows of a `DataFrame` instance can be accessed using the `loc` indexer attribute. Indexing this attribute also results in a `Series` object, which corresponds to a row of the original data frame.

```
In [30]: df.loc["Stockholm"]
Out[30]: Population    909976
         State         Sweden
         Name: Stockholm, dtype: object
```

Passing a list of row labels to the `loc` indexer results in a new `DataFrame` that is a subset of the original `DataFrame`, containing only the selected rows.

```
In [31]: df.loc[["Paris", "Rome"]]
Out[31]:
```

	Population	State
Paris	2273305	France
Rome	2872086	Italy

The `loc` indexer can also select both rows and columns simultaneously by passing a row label (or a list thereof) and a column label (or a list thereof). The result is a `DataFrame`, a `Series`, or an element value, depending on the number of selected columns and rows.

```
In [32]: df.loc[["Paris", "Rome"], "Population"]
Out[32]: Paris    2273305
         Rome     2872086
         Name: Population, dtype: int64
```

We can compute descriptive statistics using the same methods used for `Series` objects. When invoking those methods (`mean`, `std`, `median`, `min`, `max`, etc.) for a `DataFrame`, the calculation is performed for each column with numerical data types.

```
In [33]: df.mean()
Out[33]: Population    3667653.25
         dtype: float64
```

In this case, only one of the two columns has a numerical data type (the one named `Population`). Using the `DataFrame` `info` method and the `dtypes` attribute, we can obtain a summary of the content in a `DataFrame` and the data types of each column.

```
In [34]: df.info()
<class 'pandas.core.frame.DataFrame'>
Index: 4 entries, Stockholm to Paris
Data columns (total 2 columns):
Population    4 non-null int64
State         4 non-null object
dtypes: int64(1), object(1)
memory usage: 96.0+ bytes
In [35]: df.dtypes
Out[35]: Population     int64
         State         object
         dtype: object
```

The real advantages of using `pandas` emerge when dealing with larger and more complex datasets than the examples presented so far. Such data can rarely be defined as explicit lists or dictionaries, which can be passed to the `DataFrame` initializer. A more realistic situation is that the data must be read from a file or other external sources. The `pandas` library supports numerous methods for reading data from files of different formats. Here, use the `read_csv` function to read in data and create a `DataFrame` object from a CSV file.[2] This function accepts many optional arguments for tuning its behavior. See the docstring `help(pd.read_csv)` for details. Some of the most useful arguments are `header` (specifies which row, if any, contains a header with column names), `skiprows` (number of rows to skip before starting to read data, or a list of line numbers to skip), `delimiter` (the character that is used as a delimiter between column values), `encoding` (the name of the encoding used in the file, e.g., `utf-8`), and `nrows` (number of rows to read). The first and only mandatory argument to the `pd.read_csv` function is a filename or a URL to the data source. For example, to read a dataset stored in a file called `european_cities.csv`,[3] of which the first five lines are shown in the following code, we can call `pd.read_csv("european_cities.csv")`, since the default delimiter is `","` and the header is by default taken from the first line. However, we could also write out all these options explicitly.

```
In [36]: !head -n 5 european_cities.csv
Rank,City,State,Population,Date of census
1,London, United Kingdom,"8,615,246",1 June 2014
2,Berlin, Germany,"3,437,916",31 May 2014
3,Madrid, Spain,"3,165,235",1 January 2014
4,Rome, Italy,"2,872,086",30 September 2014
In [37]: df_pop = pd.read_csv("european_cities.csv",
    ...:                      delimiter=",", encoding="utf-8", header=0)
```

[2] CSV, or comma-separated values, is a common text format where rows are stored in lines and columns are separated by a comma (or some other text delimiter). See Chapter 18 for more details about this and other file formats.

[3] This dataset was obtained from the Wiki page: http://en.wikipedia.org/wiki/Largest_cities_of_the_European_Union_by_population_within_city_limits

This dataset is similar to the example data used earlier in this chapter, but now there are additional columns and many more rows for other cities. Once a dataset is read into a `DataFrame` object, it is useful to start by inspecting the summary given by the `info` method to begin forming an idea of the dataset's properties.

```
In [38]: df_pop.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 105 entries, 0 to 104
Data columns (total 5 columns):
Rank              105 non-null int64
City              105 non-null object
State             105 non-null object
Population        105 non-null object
Date of census    105 non-null object
dtypes: int64(1), object(4) memory usage: 4.9+ KB
```

Here, there are 105 rows in this dataset and that it has five columns. Only the `Rank` column is of a numerical data type. In particular, the `Population` column is not yet of numeric data type because its values are of the format `"8,615,246"` and are therefore interpreted as string values by the `read_csv` function. It is also informative to display a tabular view of the data. However, this dataset is too large to display in full. In situations like this, the `head` and `tail` methods are handy for creating a truncated dataset containing the first few and last few rows, respectively. These functions take an optional argument specifying how many rows to include in the truncated `DataFrame`. Note also that `df.head(n)` is equivalent to `df[:n]`, where `n` is an integer.

```
In [39]: df_pop.head()
Out[39]:
```

	Rank	City	State	Population	Date of census
0	1	London	United Kingdom	8,615,246	1 June 2014
1	2	Berlin	Germany	3,437,916	31 May 2014
2	3	Madrid	Spain	3,165,235	1 January 2014
3	4	Rome	Italy	2,872,086	30 September 2014
4	5	Paris	France	2,273,305	1 January 2013

Displaying a truncated `DataFrame` gives a good idea of what the data looks like and what remains to be done before the data is ready for analysis. It is common to transform columns in one way or another and reorder the table by sorting by a specific column or by ordering the index. The following explores some methods for modifying `DataFrame` objects. First, we can create new columns and update columns in a `DataFrame` simply by assigning a `Series` object to the `DataFrame` indexed by the column name, and we can delete columns using the Python `del` keyword.

The `apply` method is a powerful tool to transform the content in a column. It creates and returns a new `Series` object for which a function passed to `apply` has been applied to each element in the original column. For example, we can use the `apply` method to transform the elements in the `Population` column from strings to integers by passing a lambda function that removes the `","` characters from the strings and casts the results to an integer. Here, we assign the transformed column to a new one named `NumericPopulation`. Using the same method, tidy up the `State` values by removing extra white spaces in its elements using the string method `strip`.

```
In [40]: df_pop["NumericPopulation"] = df_pop.Population.apply(
    ...:     lambda x: int(x.replace(",", "")))
In [41]: df_pop["State"].values[:3]  # contains extra white spaces
Out[41]: array([' United Kingdom', ' Germany', ' Spain'], dtype=object)
In [42]: df_pop["State"] = df_pop["State"].apply(lambda x: x.strip())
In [43]: df_pop.head()
Out[43]:
```

	Rank	City	State	Population	Date of census	NumericPopulation
0	1	London	United Kingdom	8,615,246	1 June 2014	8615246
1	2	Berlin	Germany	3,437,916	31 May 2014	3437916
2	3	Madrid	Spain	3,165,235	1 January 2014	3165235
3	4	Rome	Italy	2,872,086	30 September 2014	2872086
4	5	Paris	France	2,273,305	1 January 2013	2273305

Inspecting the data types of the columns in the updated `DataFrame` confirms that the new column `NumericPopulation` is indeed of integer type (while the `Population` column is unchanged).

```
In [44]: df_pop.dtypes
Out[44]: Rank                int64
         City               object
         State              object
         Population         object
         Date of census     object
         NumericPopulation   int64
         dtype: object
```

We may also need to change the index to one of the columns of the `DataFrame`. For example, we could use the `City` column as an index in the current case. We can accomplish this using the `set_index` method, which takes as an argument the name of the column to use as an index. The result is a new `DataFrame` object, and the original `DataFrame` is unchanged. Furthermore, using the `sort_index` method, we can sort the data frame with respect to the index.

```
In [45]: df_pop2 = df_pop.set_index("City")
In [46]: df_pop2 = df_pop2.sort_index()
In [47]: df_pop2.head()
Out[47]:
```

City	Rank	State	Population	Date of census	NumericPopulation
Aarhus	92	Denmark	326,676	1 October 2014	326676
Alicante	86	Spain	334,678	1 January 2012	334678
Amsterdam	23	Netherlands	813,562	31 May 2014	813562
Antwerp	59	Belgium	510,610	1 January 2014	510610
Athens	34	Greece	664,046	24 May 2011	664046

The `sort_index` method also accepts a list of column names, which creates a hierarchical index. A hierarchical index uses tuples of index labels to address rows in the data frame. We can use the `sort_index` method with the integer-valued argument `level` to sort the rows in a `DataFrame` according to the nth level of the hierarchical index, where `level=n`. The following example creates a hierarchical index with `State` and `City` as indices and uses the `sort_index` method to sort by the first index (`State`).

```
In [48]: df_pop3 = df_pop.set_index(["State", "City"]).sort_index(level=0)
In [49]: df_pop3.head(7)
Out[49]:
```

State	City	Rank	Population	Date of census
Austria	Vienna	7	1794770	1 January 2015
Belgium	Antwerp	59	510610	1 January 2014
	Brussels	16	1175831	1 January 2014
Bulgaria	Plovdiv	84	341041	31 December 2013
	Sofia	14	1291895	14 December 2014
	Varna	85	335819	31 December 2013
Croatia	Zagreb	24	790017	31 March 2011

A `DateFrame` with a hierarchical index can be partially indexed using only its zeroth-level index (`df3.loc["Sweden"]`) or completely indexed using a tuple of all hierarchical indices (`df3.loc[("Sweden", "Gothenburg")]`).

```
In [50]: df_pop3.loc["Sweden"]
Out[50]:
In [51]: df_pop3.loc[("Sweden", "Gothenburg")]
Out[51]: Rank                            53
         Population                 528,014
         Date of census       31 March 2013
         NumericPopulation           528014
         Name: (Sweden, Gothenburg), dtype: object
```

City	Rank	Population	Date of census	NumericPopulation
Gothenburg	53	528,014	31 March 2013	528014
Malmö	102	309,105	31 March 2013	309105
Stockholm	20	909,976	31 January 2014	909976

If we want to sort by a column rather than the index, we can use the `sort_values` method. It takes a column name, or a list of column names, with respect to which the `DataFrame` is to be sorted. It also accepts the keyword argument `ascending`, which is a Boolean or a list of Boolean values that specifies whether the corresponding column is to be sorted in ascending or descending order.

```
In [52]: df_pop.set_index("City").sort_values(["State", "NumericPopulation"],
    ...:                                       ascending=[False, True]).head()
Out[52]:
```

City	Rank	State	Population	Date of census	NumericPopulation
Nottingham	103	United Kingdom	308,735	30 June 2012	308735
Wirral	97	United Kingdom	320,229	30 June 2012	320229
Coventry	94	United Kingdom	323,132	30 June 2012	323132
Wakefield	91	United Kingdom	327,627	30 June 2012	327627
Leicester	87	United Kingdom	331,606	30 June 2012	331606

With categorical data such as the `State` column, it is often interesting to summarize how many values of each category a column contains. Such counts can be computed using the `value_counts` method (of the `Series` object). For example, we can use the following to count the number of cities each country has on the list of the 105 largest cities in Europe.

```
In [53]: city_counts = df_pop.State.value_counts()
In [54]: city_counts.head()
Out[54]: Germany           19
         United Kingdom    16
         Spain             13
         Poland            10
         Italy             10
         dtype: int64
```

This example shows that the state with the largest number of cities on the list is Germany, with 19 cities, followed by the United Kingdom, with 16 cities, and so on. A related question is how large the total population of all cities within a state is. To answer this type of question, we can proceed in two ways: First, we can create a hierarchical index using `State` and `City` and use the `groupby` and the `sum` methods to reduce the `DataFrame` along one of the indices. In this case, we want to sum over all entries within the index level `State`, so we can use `groupby(level="State").sum()`, which eliminates the `City` index. For presentation, also sort the resulting `DataFrame` in descending order of the column `NumericPopulation`.

```
In [55]: df_pop3 = df_pop[["State", "City", "NumericPopulation"]].set_index(
    ...:     ["State", "City"])
In [56]: df_pop4 = df_pop3.groupby(level="State").sum().sort_values(
    ...:     "NumericPopulation", ascending=False)
In [57]: df_pop4.head()
Out[57]:
```

State	NumericPopulation
United Kingdom	16011877
Germany	15119548
Spain	10041639
Italy	8764067
Poland	6267409

Second, we can obtain the same results using the `groupby` method with a column name instead of index level as an argument. It allows grouping rows of a `DataFrame` by the values of a given column and apply a reduction function on the resulting object (e.g., `sum`, `mean`, `min`, `max`, etc.). The result is a new `DataFrame` with the grouped-by column as an index. Using this method, we can compute the total population of the 105 cities, grouped by state, in the following way.

```
In [58]: df_pop5 = (df_pop[["State", "NumericPopulation"]]
    ...:                .groupby("State").sum()
    ...:                .sort_values("NumericPopulation", ascending=False))
```

Note that the columns `"State"` and `"NumericPopulation"` were selected from the data frame by indexing with a list of column names we need in the group by aggregation. The `drop` method with the keyword argument `axis=1` could also have removed a column (use `axis=0` to drop rows) from the `DataFrame`. Finally, use the `plot` method of the `Series` object to plot bar graphs for the city count and the total population. The results are shown in Figure 12-2.

```
In [59]: fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
    ...: city_counts.plot(kind='barh', ax=ax1)
    ...: ax1.set_xlabel("# cities in top 105")
    ...: df_pop5.NumericPopulation.plot(kind='barh', ax=ax2)
    ...: ax2.set_xlabel("Total pop. in top 105 cities")
```

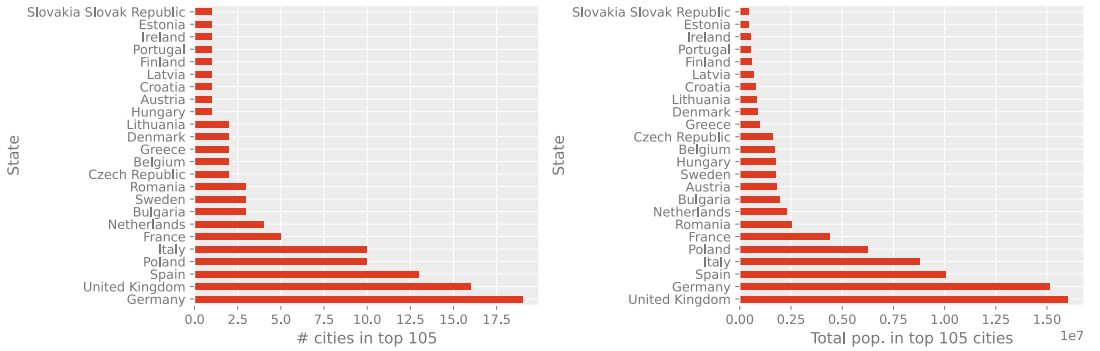

Figure 12-2. *The number of cities in the list of the top 105 most populated cities in Europe (left) and the total population in those cities (right), grouped by state*

Time Series

Time series are a common form of data in which a quantity is given, for example, at regularly or irregularly spaced timestamps or for fixed or variable time spans (periods). In `pandas`, there are dedicated data structures for representing these types of data. `Series` and `DataFrame` can have both columns and indices with data types describing timestamps and time spans. When dealing with temporal data, it is particularly useful to index the data with time data types. Using `pandas` time-series indexers, `DatetimeIndex` and `PeriodIndex`, we can carry out many common date, time, period, and calendar operations, such as selecting time ranges and shifting and resampling the data points in a time series.

To generate a sequence of dates that can be used as an index in a `pandas Series` or `DataFrame` objects, we can, for example, use the `date_range` function. It takes the starting point as a date and time string (or a `datetime` object from the Python standard library) as a first argument, and the number of elements in the range can be set using the `periods` keyword argument.

```
In [60]: pd.date_range("2015-1-1", periods=31)
Out[60]: <class 'pandas.tseries.index.DatetimeIndex'>
         [2015-01-01, ..., 2015-01-31]
         Length: 31, Freq: D, Timezone: None
```

To specify the frequency of the timestamps (which defaults to one day), we can use the `freq` keyword argument, and instead of using `periods` to specify the number of points, we can give both starting and ending points as date and time strings (or `datetime` objects) as the first and second arguments. For example, we can use the following to generate hourly timestamps between 00:00 and 12:00 on 2015-01-01.

```
In [61]: pd.date_range("2015-1-1 00:00", "2015-1-1 12:00", freq="H")
Out[61]: <class 'pandas.tseries.index.DatetimeIndex'>
         [2015-01-01 00:00:00, ..., 2015-01-01 12:00:00]
         Length: 13, Freq: H, Timezone: None
```

The `date_range` function returns an instance of `DatetimeIndex`, which can be used, for example, as an index for a `Series` or `DataFrame` object.

```
In [62]: ts1 = pd.Series(np.arange(31),
    ...:                 index=pd.date_range("2015-1-1", periods=31))
In [63]: ts1.head()
Out[63]: 2015-01-01    0
         2015-01-02    1
         2015-01-03    2
         2015-01-04    3
         2015-01-05    4
         Freq: D, dtype: int64
```

The elements of a `DatetimeIndex` object can, for example, be accessed using indexing with date and time strings. An element in a `DatetimeIndex` is of the type `Timestamp`, whivch is a `pandas` object that extends the standard Python `datetime` object (see the `datetime` module in the Python standard library).

```
In [64]: ts1["2015-1-3"]
Out[64]: 2
In [65]: ts1.index[2]
Out[65]: Timestamp('2015-01-03 00:00:00', offset='D')
```

In many aspects, a `Timestamp` and `datetime` object are interchangeable, and the `Timestamp` class has, like the `datetime` class, attributes for accessing time fields such as `year`, `month`, `day`, `hour`, `minute`, and so on. However, a notable difference between `Timestamp` and `datetime` is that `Timestamp` stores a timestamp with nanosecond resolution, while a `datetime` object only uses microsecond resolution.

```
In [66]: ts1.index[2].year, ts1.index[2].month, ts1.index[2].day
Out[66]: (2015, 1, 3)
In [67]: ts1.index[2].nanosecond
Out[67]: 0
```

We can convert a `Timestamp` object to a standard Python `datetime` object using the `to_pydatetime` method.

```
In [68]: ts1.index[2].to_pydatetime()
Out[68]: datetime.datetime(2015, 1, 3, 0, 0)
```

We can use a list of `datetime` objects to create a `pandas` time series.

```
In [69]: import datetime
In [70]: ts2 = pd.Series(
    ...:     np.random.rand(2),
    ...:     index=[datetime.datetime(2015, 1, 1), datetime.datetime(2015, 2, 1)])
In [71]: ts2
Out[71]: 2015-01-01    0.683801
         2015-02-01    0.916209
         dtype: float64
```

Data that is defined for sequences of time spans can be represented using `Series` and `DataFrame` objects that are indexed using the `PeriodIndex` class. We can construct an instance of the `PeriodIndex` class explicitly by passing a list of `Period` objects and then specify it as an index when creating a `Series` or `DataFrame` object.

```
In [72]: periods = pd.PeriodIndex([pd.Period('2015-01'),
    ...:                           pd.Period('2015-02'),
    ...:                           pd.Period('2015-03')])
In [73]: ts3 = pd.Series(np.random.rand(3), index=periods)
In [74]: ts3
Out[74]: 2015-01    0.969817
         2015-02    0.086097
         2015-03    0.016567
         Freq: M, dtype: float64
In [75]: ts3.index
Out[75]: <class 'pandas.tseries.period.PeriodIndex'>
         [2015-01, ..., 2015-03]
         Length: 3, Freq: M
```

We can also convert a `Series` or `DataFrame` object indexed by a `DatetimeIndex` object to a `PeriodIndex` using the `to_period` method (which takes an argument that specifies the period frequency, here `'M'` for month).

```
In [76]: ts2.to_period('M')
Out[76]: 2015-01    0.683801
         2015-02    0.916209
         Freq: M, dtype: float64
```

The remaining part of this section explores select features of `pandas` time series through examples. Let's look at manipulating two-time series containing temperature measurement sequences at given timestamps. There is one dataset for an indoor temperature sensor and one for an outdoor temperature sensor, with observations approximately every 10 minutes during most of 2014. The two data files, `temperature_indoor_2014.tsv` and `temperature_outdoor_2014.tsv`, are TSV (tab-separated values, a variant of the CSV format) files with two columns: the first column contains Unix timestamps (seconds since Jan 1, 1970), and the second column is the measured temperature in degree Celsius. For example, the following are the first five lines in the outdoor dataset.

```
In [77]: !head -n 5 temperature_outdoor_2014.tsv
1388530986    4.380000
1388531586    4.250000
1388532187    4.190000
1388532787    4.060000
1388533388    4.060000
```

We can read the data files using `read_csv` by specifying that the delimiter between columns is the TAB character: `delimiter="\t"`. When reading the two files, we also explicitly specify the column names using the `names` keyword argument since the files in this example do not have header lines with the column names.

```
In [78]: df1 = pd.read_csv('temperature_outdoor_2014.tsv', delimiter="\t",
    ...:                   names=["time", "outdoor"])
In [79]: df2 = pd.read_csv('temperature_indoor_2014.tsv', delimiter="\t",
    ...:                   names=["time", "indoor"])
```

Once we have created `DataFrame` objects for the time-series data, inspecting the data by displaying the first few lines is informative.

```
In [80]: df1.head()
Out[80]:
```

	time	outdoor
0	1388530986	4.38
1	1388531586	4.25
2	1388532187	4.19
3	1388532787	4.06
4	1388533388	4.06

The next step toward a meaningful representation of the time-series data is to convert the Unix timestamps to date and time objects using `to_datetime` with the `unit="s"` argument. Furthermore, we localize the timestamps (assigning a time zone) using `tz_localize` and convert the time zone attribute to the `Europe/Stockholm` time zone using `tz_convert`. We also set the `time` column as an index using `set_index`.

```
In [81]: df1.time = (pd.to_datetime(df1.time.values, unit="s")
    ...:                .tz_localize('UTC').tz_convert('Europe/Stockholm'))
In [82]: df1 = df1.set_index("time")
In [83]: df2.time = (pd.to_datetime(df2.time.values, unit="s")
    ...:                .tz_localize('UTC').tz_convert('Europe/Stockholm'))
In [84]: df2 = df2.set_index("time")
In [85]: df1.head()
Out[85]:
```

Time	outdoor
2014-01-01 00:03:06+01:00	4.38
2014-01-01 00:13:06+01:00	4.25
2014-01-01 00:23:07+01:00	4.19
2014-01-01 00:33:07+01:00	4.06
2014-01-01 00:43:08+01:00	4.06

Displaying the first few rows of the data frame for the outdoor temperature dataset shows that the index is a date and time object. A time series index represented as proper date and time objects (in contrast to using integers representing the Unix timestamps, for example) allows us to easily perform many time-oriented operations. Before exploring the data in more detail, let's plot the two time series to understand what the data looks like. We can use the `DataFrame.plot` method; the results are shown in Figure 12-3. Note that data is missing for a part of August. Imperfect data is a common problem, and handling missing data appropriately is an important part of the mission statement of the `pandas` library.

```
In [86]: fig, ax = plt.subplots(1, 1, figsize=(12, 4))
    ...: df1.plot(ax=ax)
    ...: df2.plot(ax=ax)
```

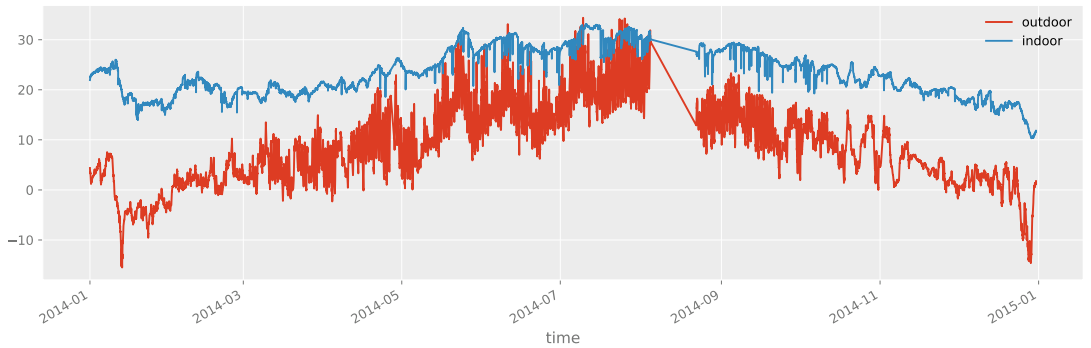

Figure 12-3. *Plot of the time series for indoor and outdoor temperature*

It is also illuminating to display the result of the `info` method of the `DataFrame` object. Doing so tells we that this dataset has nearly 50,000 data points and contains data points starting at 2014-01-01 00:03:06 and ending at 2014-12-30 23:56:35.

```
In [87]: df1.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 49548 entries, 2014-01-01 00:03:06+01:00 to 2014-12-30 23:56:35+01:00
Data columns (total 1 columns):
outdoor    49548 non-null float64
dtypes: float64(1) memory usage: 774.2 KB
```

A common operation on time series is to select and extract parts of the data. For example, from the full dataset containing all of 2014, we may be interested in selecting and analyzing only the data for January. In `pandas`, we can accomplish this in several ways. For example, we can use Boolean indexing of a `DataFrame`

to create a DataFrame for a subset of the data. To create the Boolean indexing mask that selects the data for January, we can use the pandas time-series features that allow us to compare the time-series index with string representations of a date and time. In the following code, expressions like df1.index >= "2014-1-1", where df1.index is a time DateTimeIndex instance, result in a Boolean NumPy array that can be used as a mask to select the desired elements.

```
In [88]: mask_jan = (df1.index >= "2014-1-1") & (df1.index < "2014-2-1")
In [89]: df1_jan = df1[mask_jan]
In [90]: df1_jan.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 4452 entries, 2014-01-01 00:03:06+01:00 to 2014-01-31 23:56:58+01:00
Data columns (total 1 columns):
outdoor    4452 non-null float64
dtypes: float64(1) memory usage: 69.6 KB
```

Alternatively, we can use slice syntax directly with date and time strings.

```
In [91]: df2_jan = df2["2014-1-1":"2014-1-31"]
```

The results are two DataFrame objects, df1_jan and df2_jan, containing data only for January. Plotting this subset of the original data using the plot method results in the graph shown in Figure 12-4.

```
In [92]: fig, ax = plt.subplots(1, 1, figsize=(12, 4))
    ...: df1_jan.plot(ax=ax)
    ...: df2_jan.plot(ax=ax)
```

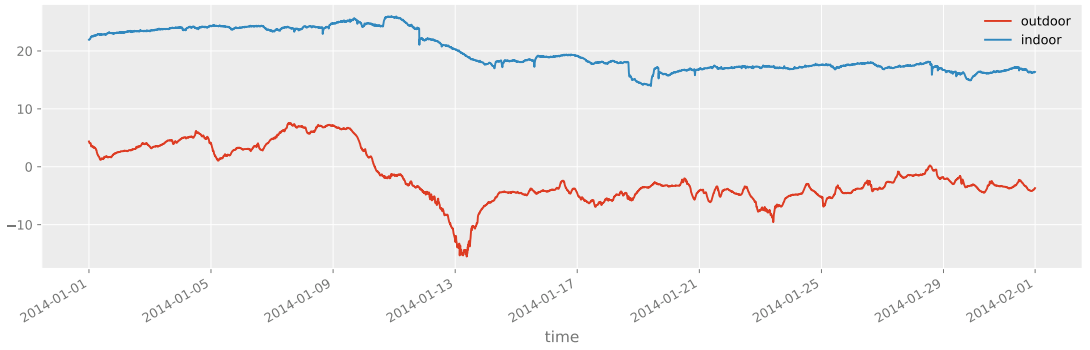

Figure 12-4. *Plot of the time series for indoor and outdoor temperature for a selected month (January)*

Like the datetime class in Python's standard library, the Timestamp class used in pandas to represent time values has attributes for accessing fields such as year, month, day, hour, minute, and so on. These fields are particularly useful when processing time series. For example, suppose we wish to calculate the average temperature for each month of the year. In this case, let's begin by creating a new column, month, which we assign to the month field of the Timestamp values of the DatetimeIndex indexer. To extract the month field from each Timestamp value, we first call reset_index to convert the index to a column in the data frame (in which case the new DataFrame object falls back to using an integer index), after which we can use the apply function on the newly created time column.[4]

[4] We can also directly use the month method of the DatetimeIndex index object, but for the sake of demonstration, a more explicit approach is used here.

```
In [93]: df1_month = df1.reset_index()
In [94]: df1_month["month"] = df1_month.time.apply(lambda x: x.month)
In [95]: df1_month.head()
Out[95]:
```

	time	outdoor	month
0	2014-01-01 00:03:06+01:00	4.38	1
1	2014-01-01 00:13:06+01:00	4.25	1
2	2014-01-01 00:23:07+01:00	4.19	1
3	2014-01-01 00:33:07+01:00	4.06	1
4	2014-01-01 00:43:08+01:00	4.06	1

Next, we can group the `DataFrame` by the new `month` field and aggregate the grouped values using the `mean` function for computing the average within each group.

```
In [96]: df1_month = df1_month[
    ...:     ["month", "outdoor"]].groupby("month").aggregate(np.mean)
In [97]: df2_month = df2.reset_index()
In [98]: df2_month["month"] = df2_month.time.apply(lambda x: x.month)
In [99]: df2_month = df2_month[
    ...:     ["month", "indoor"]].groupby("month").aggregate(np.mean)
```

After repeating the same process for the second `DataFrame` (indoor temperatures), we can combine `df1_month` and `df2_month` into a single `DataFrame` using the `join` method.

```
In [100]: df_month = df1_month.join(df2_month)
In [101]: df_month.head(3)
Out[101]:
```

time	outdoor	indoor
1	-1.776646	19.862590
2	2.231613	20.231507
3	4.615437	19.597748

We have leveraged Pandas' data processing capabilities in only a few lines of code to transform and compute with the data. There are often many ways to combine the tools provided by `pandas` to do the same or a similar analysis. For the current example, we can do the whole process in a single line of code using the `to_period` and `groupby` methods and the `concat` function (which, like `join`, combines `DataFrame` into a single `DataFrame`).

```
In [102]: df_month = pd.concat(
     ...:     [df.to_period("M").groupby(level=0).mean()
     ...:      for df in [df1, df2]], axis=1)
In [103]: df_month.head(3)
Out[103]:
```

time	outdoor	indoor
2014-01	-1.776646	19.862590
2014-02	2.231613	20.231507
2014-03	4.615437	19.597748

To visualize the results, plot the average monthly temperatures as a bar plot and a boxplot using the `DataFrame` method `plot`. The result is shown in Figure 12-5.

```
In [104]: fig, axes = plt.subplots(1, 2, figsize=(12, 4))
     ...: df_month.plot(kind='bar', ax=axes[0])
     ...: df_month.plot(kind='box', ax=axes[1])
```

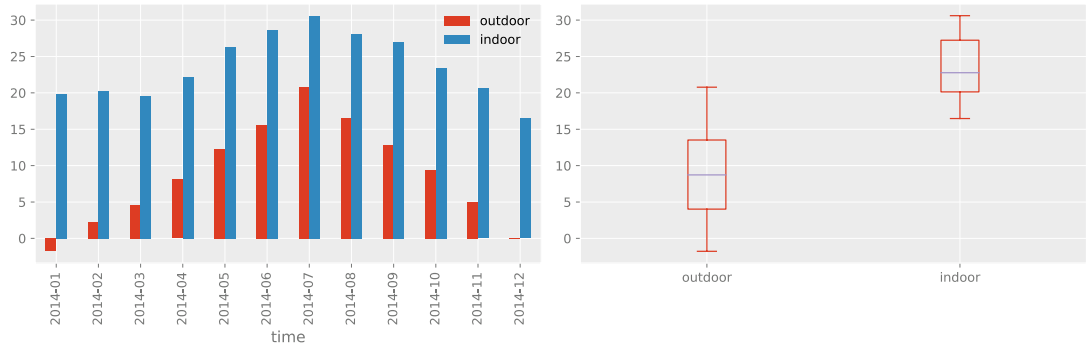

Figure 12-5. *Average indoor and outdoor temperatures per month (left) and a boxplot for monthly indoor and outdoor temperature (right)*

Finally, a handy feature of the `pandas` time-series objects is the ability to up- and down-sample the time series using the `resample` method. Resampling means changing the number of data points in a time series. It can be either increased (up-sampling) or decreased (down-sampling). For up-sampling, we need to choose a method for filling in the missing values, and for down-sampling, we need to select a method for aggregating multiple sample points between each new sample point. The `resample` method expects a string that specifies the new period of data points in the resampled time series as the first argument. For example, the string `H` represents one hour, the string `D` one day, the string `M` one month, and so on.[5] We can combine these in simple expressions, such as `7D`, which denotes seven days. The `resample` method returns a resampler object for which we can invoke aggregation methods, such as `mean` and `sum`, to obtain the resampled data.

To illustrate using the `resample` method, consider the previous two-time series with temperature data. The original sampling frequency is roughly 10 minutes, which amounts to many data points over a year. For plotting purposes or to compare the two-time series sampled at slightly different timestamps, it is often necessary to downsample the original data. This can give less busy graphs and regularly spaced time series that can be readily compared to each other. The following code resamples the outdoor temperature time series to four different sampling frequencies and plots the resulting time series. We also resample the outdoor and indoor time series to daily averages that we subtract to obtain the daily average temperature difference between indoors and outdoors throughout the year (see Figure 12-6). These types of manipulations are convenient when dealing with time series, and it is one of the many areas in which the `pandas` library shines.

[5] There are a large number of available time-unit codes. For more information, see the "Offset aliases" and "Anchored offsets" sections in the Pandas reference manual.

```
In [105]: df1_hour = df1.resample("H").mean()
In [106]: df1_hour.columns = ["outdoor (hourly avg.)"]
In [107]: df1_day = df1.resample("D").mean()
In [108]: df1_day.columns = ["outdoor (daily avg.)"]
In [109]: df1_week = df1.resample("7D").mean()
In [110]: df1_week.columns = ["outdoor (weekly avg.)"]
In [111]: df1_month = df1.resample("M").mean()
In [112]: df1_month.columns = ["outdoor (monthly avg.)"]
In [113]: df_diff = (df1.resample("D").mean().outdoor -
     ...:            df2.resample("D").mean().indoor)
In [114]: fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 6))
     ...: df1_hour.plot(ax=ax1, alpha=0.25)
     ...: df1_day.plot(ax=ax1)
     ...: df1_week.plot(ax=ax1)
     ...: df1_month.plot(ax=ax1)
     ...: df_diff.plot(ax=ax2)
     ...: ax2.set_title("temperature difference between outdoor and indoor")
     ...: fig.tight_layout()
```

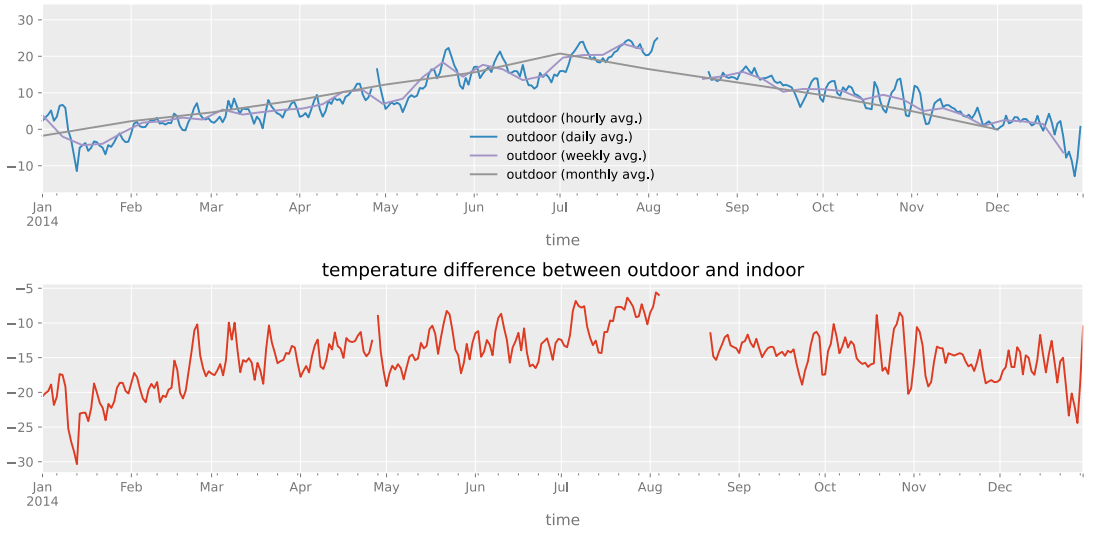

Figure 12-6. *Outdoor temperature, resampled to hourly, daily, weekly, and monthly averages (top). Daily temperature difference between outdoors and indoors (bottom)*

As an illustration of up-sampling, consider the following example, which resamples the data frame `df1` to a sampling frequency of 5 minutes, using three different aggregation methods (`mean`, `ffill` for forward-fill, and `bfill` for back-fill). The original sample frequency is approximately 10 minutes, so this resampling is up-sampling. The result is three new data frames combined into a single `DataFrame` object using the `concat` function. The first five rows in the data frame are also shown in the following example. Note that every second data point is a new sample point, and depending on the value of the aggregation method, those values are filled (or not) according to the specified strategies. When no fill strategy is selected, the corresponding values are marked as missing using the `NaN` value.

```
In [115]: pd.concat(
     ...:     [df1.resample("5min").mean().rename(columns={"outdoor": 'None'}),
     ...:      df1.resample("5min").ffill().rename(columns={"outdoor": 'ffill'}),
     ...:      df1.resample("5min").bfill().rename(columns={"outdoor": 'bfill'})],
     ...:     axis=1).head()
Out[115]:
```

time	None	ffill	bfill
2014-01-01 00:00:00+01:00	4.38	4.38	4.38
2014-01-01 00:05:00+01:00	NaN	4.38	4.25
2014-01-01 00:10:00+01:00	4.25	4.25	4.25
2014-01-01 00:15:00+01:00	NaN	4.25	4.19
2014-01-01 00:20:00+01:00	4.19	4.19	4.19

The Seaborn Graphics Library

The Seaborn graphics library is built on top of Matplotlib, and it provides functions for generating graphs that are useful when working with statistics and data analysis, including distribution plots, kernel-density plots, joint distribution plots, factor plots, heatmaps, facet plots, and several ways of visualizing regressions. It also provides methods for coloring data in graphs and numerous well-crafted color palettes. The Seaborn library is created with close attention to the aesthetics of the graphs it produces, and the graphs generated by the library tend to be both good-looking and informative. The Seaborn library distinguishes itself from the underlying Matplotlib library in that it provides polished higher-level graph functions for a specific application domain, namely, statistical analysis and data visualization. The ease with which standard statistical graphs can be generated with the library makes it a valuable tool in exploratory data analysis.

To start using the Seaborn library, set a style for the graphs it produces using the `sns.set` function. Let's work with the style called `darkgrid`, which produces graphs with a gray background (also try the `whitegrid` style).

```
In [116]: sns.set(style="darkgrid")
```

Importing `seaborn` and setting a style for the library alters the default settings for how Matplotlib graphs appear, including graphs produced by the `pandas` library. For example, consider the following plot of the previously used indoor and outdoor temperature time series. The resulting graph is shown in Figure 12-7, and although the graph was produced using the `pandas DataFrame` method `plot`, using the `sns.set` function has changed the graph's appearance (compare with Figure 12-3).

```
In [117]: df1 = pd.read_csv('temperature_outdoor_2014.tsv', delimiter="\t",
     ...:                    names=["time", "outdoor"])
     ...: df1.time = (pd.to_datetime(df1.time.values, unit="s")
     ...:               .tz_localize('UTC').tz_convert('Europe/Stockholm'))
     ...: df1 = df1.set_index("time").resample("10min").mean()
In [118]: df2 = pd.read_csv('temperature_indoor_2014.tsv', delimiter="\t",
     ...:                    names=["time", "indoor"])
```

```
     ...: df2.time = (pd.to_datetime(df2.time.values, unit="s")
     ...:                .tz_localize('UTC').tz_convert('Europe/Stockholm'))
     ...: df2 = df2.set_index("time").resample("10min").mean()
In [119]: df_temp = pd.concat([df1, df2], axis=1)
In [120]: fig, ax = plt.subplots(1, 1, figsize=(8, 4))
     ...: df_temp.resample("D").mean().plot(y=["outdoor", "indoor"], ax=ax)
```

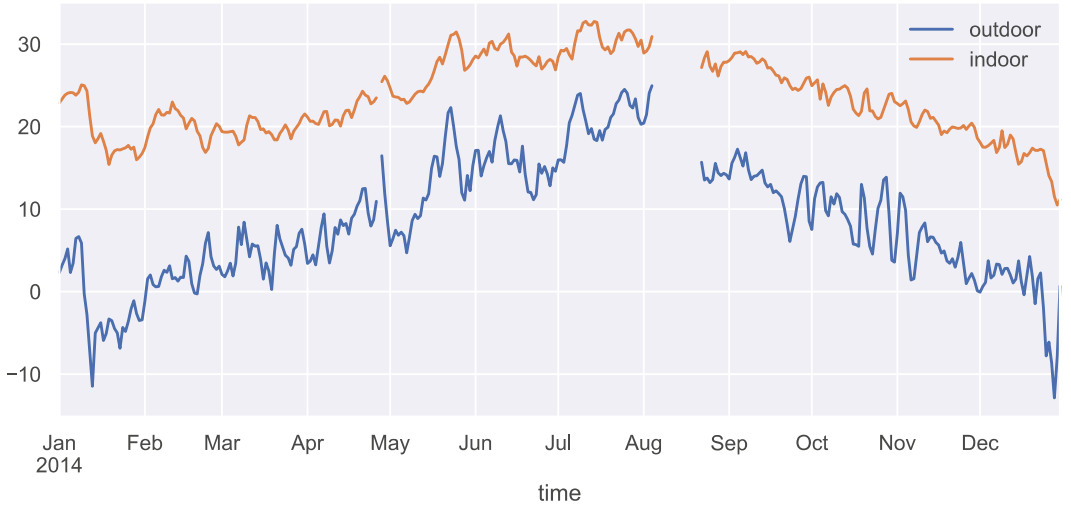

Figure 12-7. *Time-series plot produced by Matplotlib using the Pandas library, with a plot style that the Seaborn library sets up*

Apart from generating good-looking graphics, the Seaborn library's main strength is its collection of easy-to-use statistical plots. Examples are the `kdeplot` and `histplot`, which plot a kernel-density estimate plot and a histogram plot with a kernel-density estimate overlaid on top of the histogram. For example, the following two lines of code produce the graph shown in Figure 12-8. The solid blue and green lines in this figure are the kernel-density estimates that can also be graphed separately using the `kdeplot` function (not shown here).

```
In [121]: sns.histplot(
     ...:     df_temp.to_period("M")["outdoor"]["2014-04"].dropna().values,
     ...:     bins=50, kde=True)
     ...: sns.histplot(
     ...:     df_temp.to_period("M")["indoor"]["2014-04"].dropna().values,
     ...:     bins=50, kde=True)
```

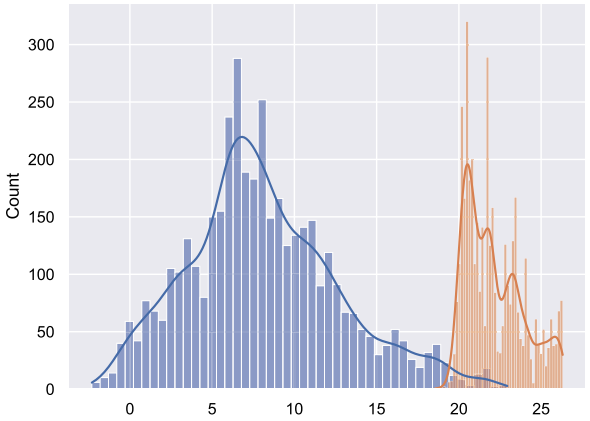

Figure 12-8. *The histogram (bars) and kernel-density plots (solid lines) for the subset of the indoor and outdoor datasets that correspond to April*

The `kdeplot` function can also operate on two-dimensional data, showing a contour graph of the joint kernel-density estimate. We can use the `jointplot` function to plot the joint distribution for two separate datasets. The following example uses the `kdeplot` and `jointplot` to show the correlation between the indoor and outdoor data series, which are resampled to hourly averages before visualized. (Missing values are dropped using the `dropna` method, since the functions from the `seaborn` module do not accept arrays with missing data.) The results are shown in Figure 12-9.

```
In [122]: sns.kdeplot(
     ...:     x=df_temp.resample("H").mean()["outdoor"].dropna().values,
     ...:     y=df_temp.resample("H").mean()["indoor"].dropna().values,
     ...:     fill=False)
In [123]: with sns.axes_style("white"):
     ...:     sns.jointplot(x=df_temp.resample("H").mean()["outdoor"].values,
     ...:                   y=df_temp.resample("H").mean()["indoor"].values,
     ...:                   kind="hex")
```

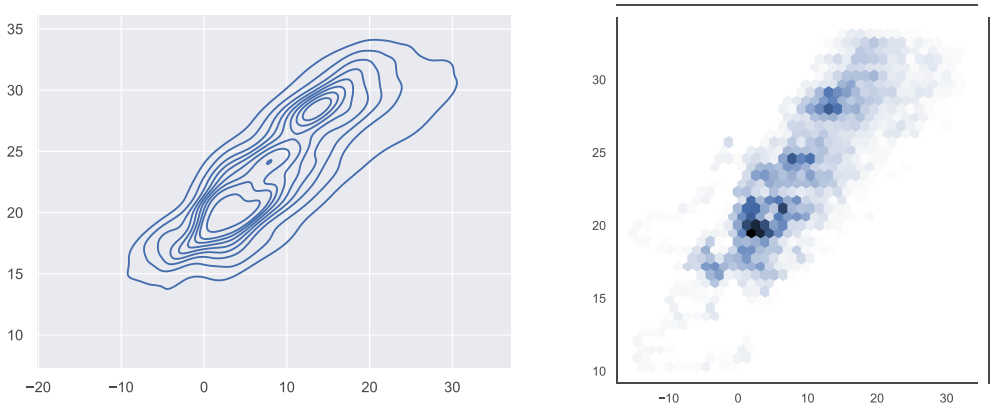

Figure 12-9. *Two-dimensional kernel-density estimate contours (left) and the joint distribution for the indoor and outdoor temperature datasets (right). The outdoor temperatures are shown on the x axis, and the indoor temperatures are on the y axis*

The `seaborn` library also provides functions for working with categorical data. A simple example of a graph type often useful for datasets with categorical variables is the standard boxplot for visualizing a dataset's descriptive statistics (min, max, median, and quartiles). An interesting twist on the standard boxplot is the violin plot, in which the kernel-density estimate is shown in the width of the boxplot. The `boxplot` and `violinplot` functions can produce such graphs, as shown in the following example, and the resulting graph is shown in Figure 12-10.

```
In [124]: fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4))
     ...: sns.boxplot(df_temp.dropna(), ax=ax1, palette="pastel")
     ...: sns.violinplot(df_temp.dropna(), ax=ax2, palette="pastel")
```

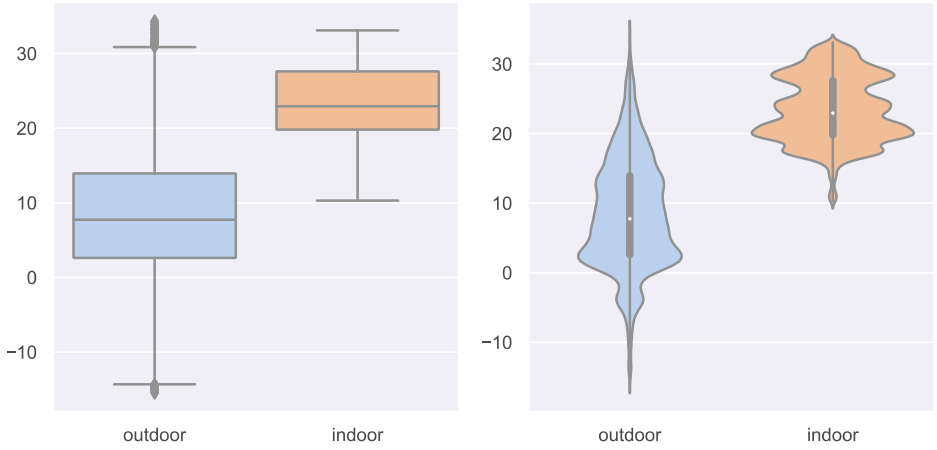

Figure 12-10. *A boxplot (left) and violin plot (right) for the indoor and outdoor temperature datasets*

As a further example of violin plots, consider the outdoor temperature dataset partitioned by the month, which can be produced by passing the month field of the data frame index as a second argument (used to group the data into categories). The resulting graph, shown in Figure 12-11, provides a compact and informative visualization of the distribution of temperatures for each month of the year.

```
In [125]: sns.violinplot(x=df_temp.dropna().index.month,
     ...:                y=df_temp.dropna().outdoor, color="skyblue")
```

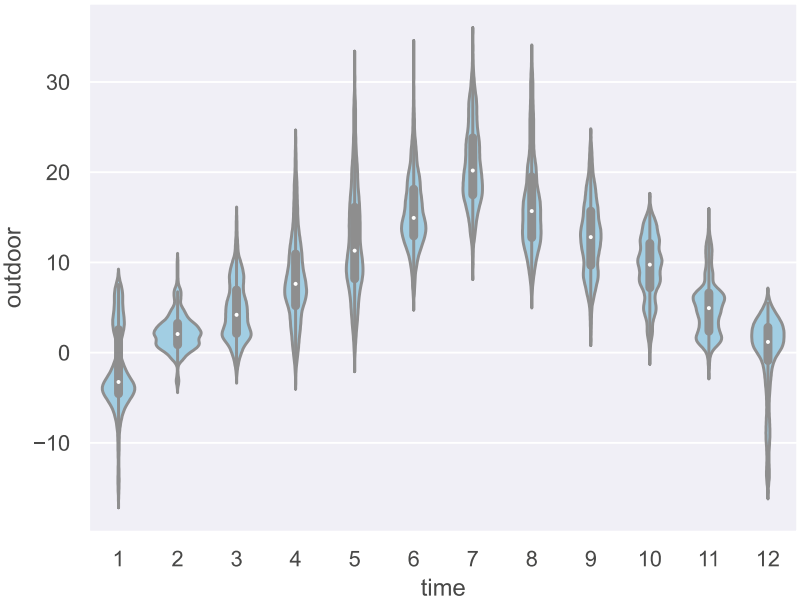

Figure 12-11. *Violin plot for the outdoor temperature grouped by month*

Heatmaps are another type of graph that is handy when dealing with categorical variables, especially for variables with many categories. The Seaborn library provides the `heatmap` function for generating this type of graph. For example, working with the outdoor temperature dataset, we can create two categorical columns, `month` and `hour`, by extracting those fields from the index and assigning them to new columns in the data field. Next, we can use the `pivot_table` function in `pandas` to pivot the columns into a table (matrix) where two selected categorical variables constitute the new index and columns. Here, let's pivot the temperature dataset so that the hours of the day are the columns and the months of the year are the rows (`index`). To aggregate the multiple data points that fall within each hour-month category, use `aggfunc=np.mean` argument to compute the mean of all the values.

```
In [126]: df_temp["month"] = df_temp.index.month
     ...: df_temp["hour"] = df_temp.index.hour
In [127]: table = pd.pivot_table(
     ...:     df_temp, values='outdoor', index=['month'], columns=['hour'],
     ...:     aggfunc=np.mean)
```

Once we have created a pivot table, we can visualize it as a heatmap using the heatmap function in Seaborn. The result is shown in Figure 12-12.

```
In [128]: fig, ax = plt.subplots(1, 1, figsize=(8, 4))
     ...: sns.heatmap(table, ax=ax)
```

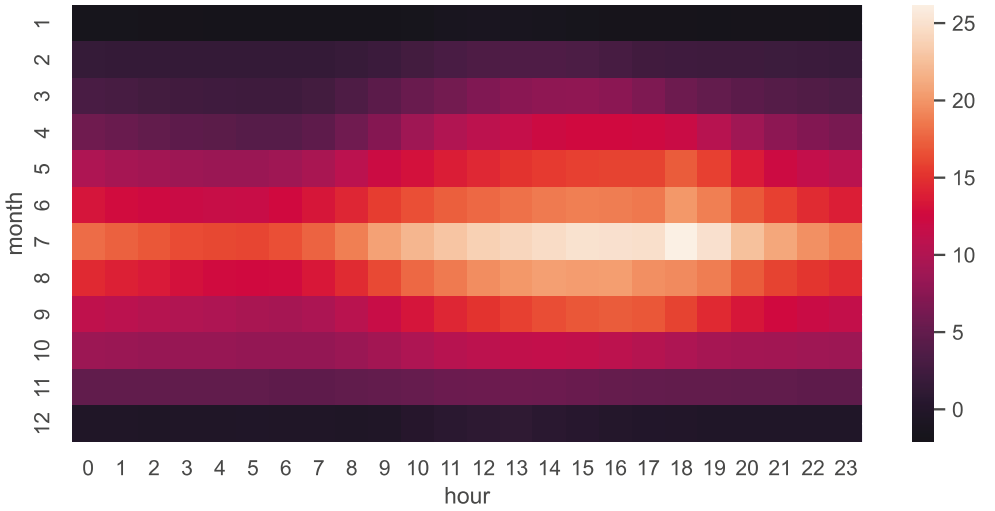

Figure 12-12. *A heatmap of the outdoor temperature data grouped by the hour of the day and month of the year*

The Seaborn library contains many more statistical visualization tools than we have been able to survey here. However, I hope that looking at a few examples of what this library can do illustrates the essence of the Seaborn library—a convenient tool for statistical analysis and exploration of data that can produce many standard statistical graphs with minimal effort. The upcoming chapters demonstrate further examples of applications of the Seaborn library.

Summary

This chapter explored data representation and processing using the Pandas library and briefly surveyed the statistical graphics tools provided by the Seaborn visualization library. The Pandas library provides the backend of much data wrangling done with Python. It achieves this by adding a higher-level abstraction layer in the data representation on top of NumPy arrays, with additional methods for operating on the underlying data. The ease with which data can be loaded, transformed, and manipulated makes it an invaluable part of the data processing workflow in Python. The Pandas library also contains essential functions for visualizing the data represented by its data structures. Visualizing data represented as the Pandas series and data frames quickly is an important tool in exploratory data analytics and for presentation. The Seaborn library takes this a step further and provides a rich collection of statistical graphs that can be produced often with a single line of code. Many functions in the Seaborn library can operate directly on Pandas data structures.

Further Reading

A great introduction to the Pandas library is given by the original creator of the library in *Python for Data Analysis* by W. McKinney (O'Reilly 2013), and it is also a rather detailed introduction to NumPy. The Pandas official documentation, available at `http://pandas.pydata.org/pandas-docs/stable`, provides an accessible and detailed description of the library's features. Another good online resource for learning Pandas is `http://github.com/jvns/pandas-cookbook`. For data visualization, we looked at the Seaborn Library, and it is well described in the documentation available on its website. With respect to higher-level

visualization tools, it is also worth exploring the `ggplot` library for Python, `http://ggplot.yhathq.com`, which is an implementation based on the renowned *The Grammar of Graphics* by L. Wilkinson (Springer, 2005). This library is also closely integrated with the Pandas library, providing convenient statistical visualization tools when analyzing data. For more information about visualization in Python, see *Beginning Python Visualization* by S. Vaingast (Apress, 2014).